e-Research: A Journal of Undergraduate Work

Volume 2
Number 2 *Vol 2, No 2 (2011)*

Article 4

September 2014

High Performance Computing Markov Models using Hadoop MapReduce

Matthew Shaffer

e-Research: A Journal of Undergraduate Work, Vol 2, No 2 (2011)

High Performance Computing Markov Models using Hadoop MapReduce

Matthew Shaffer

Abstract

In this paper, I will explain how I used the probability modeling tool, Markov Models, in combination with Hadoop MapReduce parallel programming platform in order to quickly and efficiently analyses documents and create a probability model of them. I will explain what Markov Models are, give a brief overview of what MapReduce is, explain why Markov models can be used for document analysis, explain my code of the modeling program, and examine the performance of various MapReduce platforms and techniques in analyzing documents.

Keywords: Probability Modeling Tools, Markov Models, Hadoop Maproduce, Cloud Computing

---

1 Introduction

For this experiment, Hadoop MapReduce was used to parallelize the process of creating Markov models. Markov Models are a widely used probability modeling tool. They model any environment that can be separated into discrete "states" where the probability of moving from one state to another depends only on the current state. Hadoop's MapReduce framework is an open source programming library that uses the techniques introduced by Google's MapReduce process in order to program computers to store and process vast amounts of data efficiently.

In this project, a program was encoded to analyses documents into a Markov model by modeling the probability of any particular word following another word. The local Chapman Hadoop node along with the Amazon Web Service Cloud computing platform were used in order to test how various MapReduce platforms perform. The results of these experiments were analyzed to give a rough overview of the efficiency of the various programs in processing data.

2 Parallel Programming Environment Analysis

MapReduce is a software framework first introduced by Google and designed to process large amounts of data by separating the data into smaller chunks and performing large numbers of small operations in parallel on the data. It has two steps. The first is to divide a data set up into groups and input into the "Map" operation. This operation works by taking the input data and output a set of "Key-Value" pairs. These pairs are generally a set of tuples in no particular order that are outputted as they are found. The next step is to organize the data by sorting the keys and aggregating the values together. This data may then be given to another map step, in the case of a multi-step MapReduce job, or then given to the "reduce" step. This step works by taking all the values for a key and summing them up into a result which is then outputted from the program.

The Hadoop MapReduce Framework is an open source version of MapReduce developed by the Apache Software Foundation[5]. The Hadoop MapReduce Framework offers two main ways of running MapReduce program: Streaming and Jar files. The streaming system works by launching an outside program and feeding in the data to the program as the standard in and reading the results from the standard out. The Data is "streamed" in and out of another program. The jar file implementation works by taking a jar file that contains a program that extends the MapReduce base and running it using the standard MapReduce API.

Streaming functions differently from the Jar file functionality in how information is moved from the "map" step to the "reduce" step. A Jar file program will sum up data into an aggregate set of data between these two steps. For Example: <Book, 1>, <Duck, 1>, <Book, 1> becomes <Book, 1 1>, <Dog, 1>. The Streaming operation simply organizes the keys but does not aggregate them. For example, the above would become: <Book, 1>, <Book, 1>, <Duck, 1>.

The Hadoop framework allows for any of its serializable datatypes to be used as the key and values in a jar file job. For example, in the program used for this paper Text objects were used as both the input and output of each step. This makes the actual job more flexible and able to process more types of programs and data.

2.1 Markov Models A brief analysis

Markov Models are a probability modeling tool. A Markov Models is any model, such as a graph, that has the Markov Property. The property applies to any system such that the system has discrete states and given the present state of the system and all its previous states the probability of the state changing from the current one to another depends only on the current state. In the case of discrete values and time indexes,

$$\mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1} \dots X_0 = x_0) = \mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1})$$ [1].

This means that at any time the probability of anything happening depends only on the present situation and not on any previous situation. This allows a model to show probability as a simple statistically likelihood of the transition between states while disregarding pervious state. This simplifies the task of predicting the likelihood of something happening by making it so that only a single model and its probabilities need to be considered at any time.

For example, for my project uses Markov models to model documents. A document can be viewed as one word following another many times over. A Markov model can be made to model this behavior by examining how often one word changes to another word. Thus each word is a state and the models finds how likely a state is to change to another state, or word here, based on the current word and not on any previous states, or words.

Often times a technique called "smoothing" is used on Markov models. This technique gives the model at least a small probability of transitioning from any one state to any other state regardless of whether or not a particular state followed another in the data set. For example, there would be a small chance for elephant to follow the word river even if the word elephant never followed river in any of the data that was used to make the model. This technique helps to simulate the randomness of an actual environment and to try and deal with situations where an event can occur but simply never did in the data set that was used to generate the model.

3 Parallel Computing Experiments

MapReduce is particularly suited for Markov modeling. Since the probability of moving to each state depends only on the current one there is no need to record what words precede the current one. This means that the data can be easily divided up into small chunks without losing any information. In addition, the idea of key-value pairs works well with the Document analysis since each word can be broken up into word-next word pairs that follow the key-value method and then the number of times a word follows another can be summed in the reduce step in the same way other aggregated values are summed up in reduce.

My Python and Java Programs are based on the example WordCount programs provided by Hadoop on their website [2]. I will explain my Python code and then explain my Java code.

MarkovMapper:
The inputted lines (from Standard in) are formatted by removing punctuation, converted the string to lower case, and removing extra white space. They are then moved into a list of individual words. This formatting is to avoid creating erroneous distinctions between capitalized letters and uncapitalize letters and words with, for example, a period at the end and those that don't have a period. This makes the words more uniform and the results more accurate.

```
for line in sys.stdin:
   line = line.strip().lower().translate(None, string.punctuation)
   words = line.split()
```

Then, the words, assuming that there are any at all, are printed (to standard out) in order as the word and then the next word in the list after it. A tab is used to distinguish the key from the value.

```
 if(len(words) > 0):
        for i in range(len(words)-1):
      print '%s\t%s' % (words[i], words[i+1])
```

MarkovReducer:
As words are read in (from Standard in), they are split using the tab that was used to distinguish keys and values in the previous step. The key-value pair is then checked against the contents of a dictionary in the program, called counter. If the key does not already exist, it is added with a value of 1 otherwise the previous value is incremented.

```
for line in sys.stdin:
   nkey, word = line.split('\t', 1)
   if (nkey, word) not in counter:
      counter[(nkey, word)] = 1
   else:
      counter[(nkey, word)] += 1
```

After each line has been inputted, the keys of the dictionary, i.e. the key-value tuple, are collected into a list. The key, followed by the value and the contents of the key-value in the dictionary, i.e. the counter for this pair, is printed with it.

```
outwords = counter.keys()
for outword in outwords:
   output += str(outword[0]) + "\t" +str(outword[1])+ " " + str(counter[outword])+
"\n";
print output
```

Thus, the mapper simply creates tuples of words and the word that follows them and the reducer sums up the number of each pair by using a dictionary and prints out the counts for each pair.

Markov.java:

Note: Both my Map and Reduce classes have been altered from using the default key-value pairs of text and integer to text and text by changing the map and reduce functions like so:

```
public static class Map
     extends Mapper<LongWritable, Text, Text, Text>{..
public static class Reduce
     extends Reducer<Text, Text, Text, Text> {...
```

My Mapper Function works by first taking each line that is inputted and stripping it of all characters except Letters and spaces and converting it lower case. Once again, this is to remove any apparent differences to the program between words that have varying cases and formats in order to make the words uniform and improve the test results.

```
String line = value.toString().replaceAll("[^A-Za-z ]","").toLowerCase();
```

The line is then split into individual strings using the StringTokenizer. If there are any lines, the Tokenizer sets the initial word into a variable called previous. It then goes through each word and sets the word it reads to be the variable "word" and prints the word along with the previous word then making the current word the previous word. In this way, it prints each word along with the word that proceed it by simply going through each word and printing the old word with the current one then moving the current one into the previous spot.

```
StringTokenizer tokenizer = new StringTokenizer(line);
      if(tokenizer.hasMoreTokens())
      {
proceed.set(tokenizer.nextToken());
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(proceed, word);
       proceed.set(word);
```

For the reducer, the program first creates an ArrayList and goes through each inputted value adding them to the Arraylist (as a String object). It then sorts it using Collections.sort(). The program then checks for the special case of only having one value in the list of words and simply creates an output of the word and the number one for this case. Otherwise, for each word, starting from index 1 it checks the word against the previous word. If the words are the same, a counter is incremented, if they are different then the word and the counter are added to an output string and the counter is reset. Once each word has been checked, the last word is added to the string (Since it will not have yet have been printed due to the fact that only different value words are printed and the last word cannot be checked as different since no words follow it). The Key along with its output string is then printed.

```
    for(i=1; i < Words.size(); i++)
    {
      if(!((String)Words.get(i)).equals((String)Words.get(i-1)))
      {
        out += (" "+(String)Words.get(i-1)+" "+Integer.toString(sum)+",");
        sum = 1;
      }
      else
      {
                sum +=1;
      }
     }
     //Grab the final word, which will not have fired by now
     out += (" "+(String)Words.get(i-1)+" "+Integer.toString(sum));
  }
  context.write(key, new Text(out));
```

A Sample output for the Streaming program looks like this:
absorbing piece 1
absorbing some 1
absorbs all 1
absorbs into 1
absorbs it 1
absorbs light 1
absorbs my 2
absorbs the 3
absorbs war 1

absorption of 4
absorption the 1
absorption was 1
absorptive wall 1
abstain from 2
abstainer and 1
abstemious to 1

Each word is matched with a value and the number of times it appears is listed after that.

A Sample output for the Java code looks like:
abridged copy 1 1
abridgements all 1 1
abridges the 1 1
abridging the 1 1
abroad and 1, waylaying 1 1
abrupt and 1 1
abruptly back 1, bent 1, his 1, i 1, the 1, what 1 1
abscission of 1 1
absconded somewhere 1 1
absence and 1, of 7, on 1, through 1 1
absent face 1, others 1 1
absentee the 1 1
absently ocularly 1 1
absentminded beggar 3, beggars 1 1

Each word is followed by the list of words that followed it along with the number of times each followed it. Unfortunately, I was not able to remove the default formatting of adding a tab and a 1 in the final output for each value. I did not have the time or expertise to find where these defaults are specified in the code and change them.

4 Performance Analysis

To test the performance of my program I tested the speed of my program with single instance set-ups for both streaming python and the custom jar on the local server and for the Custom Jar on AWS. I also tested the speed up of the Jar File on increasing numbers of instances on AWS. I used 4 data sets to test performance: a small data set of 3546 KB, a medium data set of 14400 KB, and large data set of 35248 KB, and an extra large dataset of 55712 KB (all files were obtained from the Guttenberg Project website [3]).

To test on the local server, I used the "time" command while running each program to get a measure of the total time the program ran. I ran each ten times and average the results to get the final times.

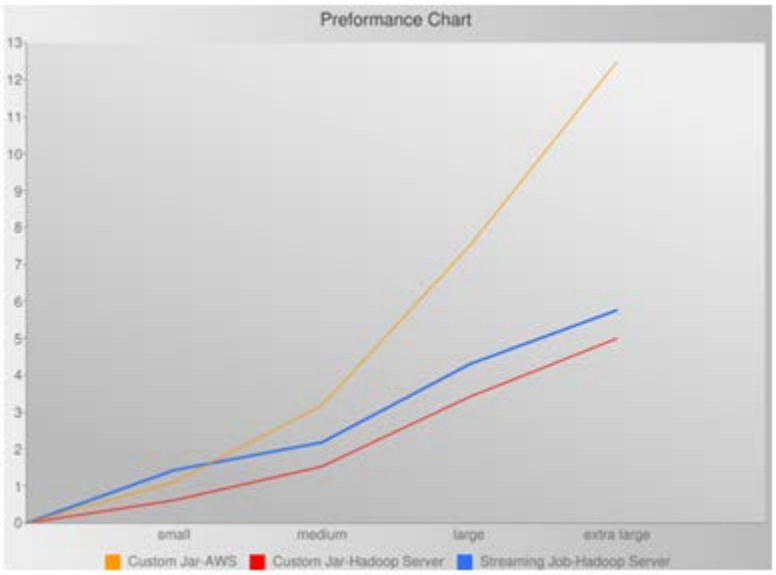
The Results are:
Streaming Python -
Small -1:26.378
Medium - 2:11.986
Large - 4:18.141
Extra Large - 5:46.468

Jar
Small - 0:37.112
Medium - 1:32.873
Large - 3:25.36
Extra Large - 4:59.918

The same data was uploaded to the Amazon Web Services, AWS, and the JAR file program was run on each dataset. Each dataset was run and recorded only 3 or 4 times due to the cost of running programs using AWS. The execution time was measured by comparing the time stamp of the first and last entry in the SysLog file of the job. Unfortunately, the streaming job fails due to unknown reasons during the shutdown phase so I was unable to get results for this type of job on the AWS.

The results are:
Small -1:07
Medium - 3:11
Large - 7:30
Extra Large - 12:30.5

I compiled the results into a chart that graphs time in minutes against data size:



The Data set sizes do not conform to a linear scale up, but, even so, it is clear from my results that on average the Jar file preformed better then the streaming job. This may be due to the overhead of the streaming of information from and to the python processes. The AWS was also fairly slow but this is likely due to the time it takes to initialize some functions on the cloud as well as the possibility that a single cloud instance has less computing power or memory then the local server.

As a further experiment, a speed-up test was run. Speed-Up is measured as the time for the data to be executed on one core divided by the time it takes for the data to be executed on *p* cores. This was tested by taking the Extra Large data set and running it on the AWS with increasing numbers of instances. Only one test result was measured for each, due to the fact that running multiple tests with large number of instance could quickly become expensive. As a result, this data is only a relative indicator.

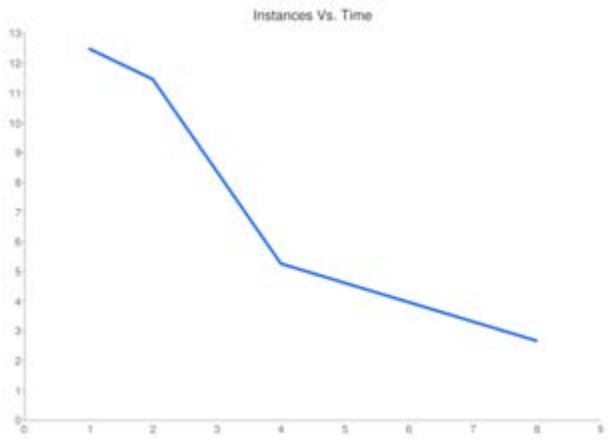
The results are:
1 instance -12:30.5
SpeedUp: 1

2 instance - 11:28
SpeedUp: 1.0908

4 instance - 5:16
SpeedUp: 2.375

8 instance - 2:40
SpeedUp: 4.6906

This chart plots the time it takes for the code to run vs. the number of instances:



In the speed-up test, the program appears to perform at roughly half of the ideal speed up. In ideal speed-up, two processors working twice as fast as one, four working four times as fast as one, and so on. This data is only a relative indicator, but points to a good speed-up but definitely one that could still be improved. Further testing and different datasets might reveal different results but this data does indicate the increasing benefit of using additional instances even if they do not perform ideally.

5 Conclusions

It is clear from my results that the Amazon web services preformed much slower than the local cluster. It is also clear that Hadoop continues to run well even when dealing with increasingly large datasets. Custom JAR files also appear to run faster overall then streaming jobs. Increasing number of instances also appear to, although not offering perfect speed up, do offer good speed up and provide a good benefit to using more processors on a job.

The process of streaming information in and out of a program seems to be a draw-back on performance speed. The transfer of information creates an additional overhead that detracts from the main process of processing the data. JAR files are thus the most effective way of using Hadoop. The speed-up performance of the system is also good but not ideal. Ideal performance is rare in any system and Hadoop does use additional cores effectively without significant loss of processing power when additional cores are added even at only half speed-up.

The AWS, although offering slower speeds here, might be hinder by potentially being slower than the local cluster overall and may be less fine-tuned to this particular type of computation. It is possible that the AWS which is a more general computing environment that that has a MapReduce environment layered onto it and must deal with sharing resources among multiple cloud users simultaneously is far more strained and has more overhead then a local cluster. This makes using a local system more efficient. The AWS is easier to use though since no equipment needs to be purchased, no system needs to be set up, and no maintenance is necessary for it. The AWS is simply an upload and go system that requires no upfront set up like a local computing cluster.

Overall, Hadoop does seem like a very efficient way to manage large datasets as even the 55 MB file was able to be processed in under 5 minutes on average with the simple 2 processor local cluster. This makes processing vast

amounts of data quicker than the conventional system which is the equivalent of the much slower one instance system in this experiment.

Further research can be made into this field by exploring the speed-up of MapReduce in other types of programs, as well as exploring the speed-up of MapReduce Streaming jobs on AWS. A comparison of both types of jobs on the AWS and local server would yield interesting data into the general efficient of using the AWS cloud system as opposed to a local host. More research can also be done into exploring the optimal number of processes to work on a job by exploring speed-up vs Data size and finding the most efficient number of processes per unit of data and exploring to see if the AWS is more efficient at certain types of computations or programs then it is at other types of computations.

References

1. http://en.wikipedia.org/wiki/Markov_property

2. http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html

3. http://www.gutenberg.org

4. http://en.wikipedia.org/wiki/Markov_chain

5. http://hadoop.apache.org/mapreduce/

6. http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html

7. http://www.velocityreviews.com/forums/t138330-remove-punctuation-from-string.html

8. http://stackoverflow.com/questions/265960/best-way-to-strip-punctuation-from-a-string-in-python

9. http://hadoop.apache.org/common/docs/r0.18.3/streaming.html

Appendix

MarkovMapper.py:

```
#!/usr/bin/env python
import sys
import string
# input comes from STDIN (standard input)
for line in sys.stdin:
   # remove leading and trailing whitespace
   line = line.strip().lower().translate(None, string.punctuation)
   # split the line into words
   words = line.split()
   if(len(words) > 0):
         for i in range(len(words)-1):
      # write the results to STDOUT (standard output);
      # what we output here will be the input for the
      # Reduce step, i.e. the input for reducer.py
      #
      # tab-delimited; the trivial word count is 1
      print '%s\t%s' % (words[i], words[i+1])
```

MarkovReducer.py:

```
#!/usr/bin/env python
from operator import itemgetter
import sys
counter = {}
# maps words to their counts
# input comes from STDIN
for line in sys.stdin:
   # remove leading and trailing whitespace
   line = line.strip()
   # parse the input we got from mapper.py
   nkey, word = line.split('\t', 1)
   if (nkey, word) not in counter:
      counter[(nkey, word)] = 1
   else:
      counter[(nkey, word)] += 1
outwords = counter.keys()
outwords.sort()
output = ""
for outword in outwords:
   output += str(outword[0]) + "\t" +str(outword[1])+ " " +
str(counter[outword])+ "\n";
print output
```

Markov.java:

```
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
 import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;
import org.apache.hadoop.util.*;
public class Markov extends Configured implements Tool {
  public static class Map
     extends Mapper<LongWritable, Text, Text, Text> {
    private Text proceed =new Text();
    private Text word = new Text();
    public void map(LongWritable key, Text value, Context context)
       throws IOException, InterruptedException {
     String line = value.toString().replaceAll("[^A-Za-z ]","").toLowerCase();
     StringTokenizer tokenizer = new StringTokenizer(line);
          if(tokenizer.hasMoreTokens())
          {
  proceed.set(tokenizer.nextToken());
       while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
   if(!word.toString().equals("") && !proceed.toString().equals("")){
        context.write(proceed, word);
             proceed.set(word);
   }
      }
          }
```

```
    }
  }
  public static class Reduce
    extends Reducer<Text, Text, Text, Text> {
   public void reduce(Text key, Iterable<Text> values,
     Context context) throws IOException, InterruptedException {
ArrayList Words = new ArrayList();
    for (Text  val : values)
{
            Words.add(val.toString());
}
Collections.sort(Words);
int sum = 1;
int i;
String out = "";
     if(Words.size() == 1)//if we only have one word, just write it
  out = ((String)Words.get(0)+" "+Integer.toString(1));
else
     {
  for(i=1; i < Words.size(); i++)
  {
    if(!((String)Words.get(i)).equals((String)Words.get(i-1)))
    {
      out += (" "+(String)Words.get(i-1)+" "+Integer.toString(sum)+",");
      sum = 1;
    }
    else
    {
              sum +=1;
    }
   }
   //Grab the final word, which will not have fired by now
   out += (" "+(String)Words.get(i-1)+" "+Integer.toString(sum));
}
context.write(key, new Text(out));
  }
 }
  public int run(String [] args) throws Exception {
   Job job = new Job(getConf());
   job.setJarByClass(Markov.class);
   job.setJobName("MarkovChain");
   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(Text.class);
   job.setMapperClass(Map.class);
   job.setCombinerClass(Reduce.class);
   job.setReducerClass(Reduce.class);
   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);
   FileInputFormat.setInputPaths(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));
   boolean success = job.waitForCompletion(true);
   return success ? 0 : 1;
  }
```

```
    public static void main(String[] args) throws Exception {
      int ret = ToolRunner.run(new Markov(), args);
      System.exit(ret);
    }
}
```

Raw Test Results:

Local Server --
Streaming Python -
Small -
1:25.688
1:25.429
1:23.742
1:25.724
1:25.855
1:25.815
1:30.044
1:29.919
1:25.844
1:25.722
Average: 1:26.378
Medium -
2:12.081
2:11.994
2:9.680
2:13.942
2:11.947
2:11.106
2:9.959
2:14.083
2:12.995
2:12.072
Average: 2:11.986
Large -
4:18.537
4:18.459
4:20.426
4:17.452
4:18.370
4:18.455
4:14.463
4:17.380
4:17.457
4:20.417
Average: 4:18.141
Extra Large -
5:47.015
5:49.884
5:46.757
5:41.848
5:44.795
5:48.738
5:46.760
5:47.276
5:47.769
5:43.842
Average:5:46.468
Jar File -
Small -
0:35.958
0:37.924
0:35.854
0:36.891
0:37.888
0:37.872
0:37.932
0:36.889
0:36.950
0:36.958
Average: 0:37.112
Medium -
1:31.184
1:33.136
1:33.159
1:33.159
1:35.271
1:33.151
1:31.167
1:33.243
1:31.156
1:32.178
Average: 1:32.873
Large -
3:25.738
3:26.654
3:24.686
3:25.567
3:24.830
3:24.602
3:24.645
3:25.629
3:26.653
3:24.597
Average: 3:25.36
Extra Large -
5:1.097
5:0.996
4:56.993
4:56.950
5:6.990
4:56.013
5:9.185
4:55.953
4:59.993
4:55.018
Average: 4:59.918
AWS --
Jar -
1 instance -
Small -
1:08
1:05
1:08
1:08
Average:1:07
Medium -
3:03
3:03
3:28
Average: 3:11
Large -
7:26
7:17
7:46
Average: 7:30
Extra Large -
11:32
12:36
12:26
13:28
Average: 12:30.5
SpeedUp: 1
2 instance -
Extra Large -
11:28
SpeedUp:1.0908
4 instance -
Extra Large -
5:16
SpeedUp:2.375
8 instance -
Extra Large -
2:40
SpeedUp:4.6906